SYSTEMS AND METHODS FOR

SOFTWARE EXTENSIBLE MULTI-PROCESSING

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims the benefit of U.S. Provisional Application Serial Number 60/459,538 titled "Method and Apparatus for an Array of Software Extensible Processors," filed March 31, 2003, which is incorporated herein by reference.

BACKGROUND

1. Field of the Invention

[0002] The present invention relates generally to processing systems, and more particularly to methods and systems for software extensible multi-processing.

2. Description of the Prior Art

[0003] Computationally intensive applications, such as modeling nuclear weaponry, simulating pharmaceutical drug interactions, predicting weather patterns, and other scientific applications, require a large amount of processing power. General computing platforms or engines have been implemented to provide the computational power to perform those applications. Such general computer computing platforms typically include multiple single-chip processors (i.e., central processor units, or "CPUs") arranged in a variety of different configurations. The number of CPU's and the interconnection topology typically defines those general computing platforms.

[0004] To improve the functionality, reduce cost, increase speed, etc. of the general computer computing platforms, the multiprocessors and their architectures are migrating onto a system-on-a-chip ("SOC"). However, these conventional approaches to designing multiprocessor architectures are focused on either the general programming environment or on a particular application. These conventional approaches, however, cannot make many assumptions about (i.e., predict) or adapt its resources to optimize computations and communications in accordance with the user's application. This deficiency exists because the number of applications varies widely and each often has requirements that vary dynamically over time, depending on the amount of resources required. Also, those approaches that are focused on one particular application often provide high performance for only one specific application and thereby are inflexible to a user's changing needs. Further, the traditional approaches do not allow a user to optimize the amount of hardware for the user's specific application, resulting in a multiprocessor architecture with superfluous resources, among other deleterious effects.

[0005] Additionally, conventional approaches do not optimize communications among processors of a multiprocessor architecture for increased speeds and/or do not easily allow scalability of the processors of such an architecture. For example, one approach provides for "cache coherency," which allows for creation of a programming model that is relatively less resource-intensive. With cache coherency, the programming model is similar to programming a uniprocessor. However, cache coherency is expensive in terms of hardware, for example, and does not scale well as the number of nodes increases. Scaling cache coherency beyond four nodes usually requires significant hardware complexity. In contrast, another approach provides for "message passing" to obtain a more scalable solution. But this message passing typically requires the users to learn a new programming model. Furthermore, message passing machines and architectures often have additional hardware overhead as each processor element must have its own copy of the program for execution.

[0006] Some multiprocessor systems have used interface protocols, such as HyperTransport from the HyperTransport Technology Consortium of Sunnyvale, California, for communications between processors. Other examples of interface protocols used are Peripheral Component Interconnect (PCI) Express and RapidIO from the RapidIO Trade Association of Austin, Texas. These interface protocols have been primarily used in high-performance processing systems such as super computers, which are very expensive. The interface protocols have also been used in general purpose processing systems. In one example, one system used Hypertransport channels in an array of Advanced Micro Devices (AMD) processors from Advanced Micro Devices, Inc. of Sunnyvale, California. These general purpose processing systems are more expensive than embedded systems because the general purpose processing systems have to include additional functionality to run a variety of applications that may change dynamically.

SUMMARY OF THE INVENTION

[0007] The invention addresses the above problems by providing systems and methods for software extensible multi-processing. A system for processing applications includes processor nodes and links interconnecting the processor nodes. Each node includes a processing element, a software extensible device, and a communication interface. The processing element executes at least one of the applications. The software extensible device provides additional instructions to a set of standard instructions for the processing element. The communication interface communicates with other processor nodes.

[0008] In some embodiments, each one of the processor nodes is on a separate chip. In other embodiments, at least some of the processor nodes are on the same chip. The processor nodes may be configured in an array. In some embodiments, the communication interface communicates with the other processor nodes using shared memory. In other embodiments, the communication interface communicates with the other processor nodes using message passing. In some embodiments, the communication interface communicates with the other processor nodes using channels between the processor nodes. In some embodiments, at least one of the processor nodes is different from the other processor nodes.

[0009] In another embodiment, a method includes the step of executing an application in at least one processing element in a plurality of processor nodes. The method also includes the step of providing an additional instruction to a set of standard instructions for the processing element using at least one software extensible device in the plurality of the processor nodes. The method also includes the step of communicating between the processor nodes using at least one communication interface in a plurality of the processor nodes.

BRIEF DESCRIPTION OF THE DRAWINGS

[00010] FIG. 1 is a diagram of a processing system in an exemplary implementation of the invention;

[00011] FIG. 2 is a diagram of a processor node in an exemplary implementation of the invention;

[00012] FIG. 3 is a block diagram of a processor node with neighboring processor nodes in an exemplary implementation of the invention;

[00013] FIG. 4 is a diagram for illustrating communication paths for an array of processing nodes in an exemplary implementation of the invention;

[00014] FIG. 5 is a block diagram for an application in an exemplary implementation of the invention;

[00015] FIG. 6 is an illustration of an example of a conceptual view of an application used in an exemplary programming model of the present invention;

[00016] FIG. 7 is a diagram of an array of processor nodes using time division multiplexing in an exemplary implementation of the invention;

[00017] FIG. 8 is a diagram of an array of processor nodes using bundling in an exemplary implementation of the invention;

[00018] FIG. 9 is a diagram of a software extensible processor chip in an exemplary implementation of the invention;

[00019] FIG. 10A is a diagram of two software extensible processor chips in an exemplary implementation of the invention;

[00020] FIG. 10B is a diagram of four software extensible processor chips in an exemplary implementation of the invention;

[00021] FIG. 10C is a diagram of a three by three array of software extensible processor chips in an exemplary implementation of the invention;

[00022] FIG. 10D is a diagram of two software extensible processor chips with a software extensible system on a chip in an exemplary implementation of the invention;

[00023] FIG. 11A is diagram of a first non-rectangular configuration of processor nodes in an exemplary implementation of the invention;

[00024] FIG. 11B is diagram of a second non-rectangular configuration of processor nodes in an exemplary implementation of the invention;

[00025] FIG. 11C is diagram of a third non-rectangular configuration of processor nodes in an exemplary implementation of the invention;

[00026] FIG. 11D is diagram of a fourth non-rectangular configuration of processor nodes in an exemplary implementation of the invention; and

[00027] FIG. 12 is a routing algorithm in an exemplary implementation of the invention.

DETAILED DESCRIPTION OF THE INVENTION

[00028] As shown in the exemplary drawings wherein like reference numerals indicate like or corresponding elements among the figures, exemplary embodiments of a system and method according to the present invention are described below in detail. It is to be understood, however, that the present invention may be embodied in various forms. Therefore, specific details disclosed herein are not to be interpreted as limiting, but rather as a basis for the claims and as a representative basis for teaching one skilled in the art to employ the present invention in virtually any appropriately detailed system, structure, method, process or manner.

Processing System – FIGS. 1-4

[00029] FIG. 1 depicts a diagram of a processing system 100 in an exemplary implementation of the invention. The processing system 100 includes Double Data Rate (DDR) memory controllers 110 and 120, Input/Output (I/O) devices 131-138, and an array of processor nodes 140. In this embodiment, the array of processor nodes 140 is a four by four array of processor nodes 150. Other embodiments comprise various combinations of numbers and different configurations of processor nodes to form a multiprocessor architecture. The multiprocessor architecture including such an array of processor nodes can be scaled to form a multiprocessor of any number of processor nodes, such as four by four processor nodes, or sixteen by sixteen processor nodes. In some embodiments, such scaling can be selected according to a particular manufacturing process on which the array of processing nodes 140 are suitable for operating upon.

[00030] In some embodiments, the array of processor nodes 140 is implemented as a multiprocessor system-on-a-chip, where multiple processor nodes 150 are integrated into a single

chip. In some embodiments, the array of processor nodes 140 is a collection of chips on a board, where each chip comprises a processor node 150. In some embodiments, some of the processor nodes are different from each other creating a heterogeneous array of processor nodes 140.

[00031] The following description is for processor node 150 but also applies to the other processing nodes in the array of processor nodes 140. The processor node 150 comprises a processing element (PE) 152 and a processor network switch 154. The processor network switch 154 is coupled to the processing element 152. The processor network switch 154 is coupled to neighboring processor network switches in other processor nodes, the DDR memory controller 110, and the I/O device 131. A description of the processor node 150 is discussed in further detail below.

[00032] FIG. 2 depicts a diagram of a processor node 200 in an exemplary implementation of the invention. The processor node 200 includes an instruction set extension fabric (ISEF) 210, a processing element 220, an instruction (INST) cache 222, a data cache 224, a double port random access memory (DP-RAM) 230, a processor network interface 240, and a processor network switch 250.

[00033] The ISEF 210 is coupled to the processing element 220. The ISEF 210 includes programmable logic for enabling application-specific instructions ("instruction extensions") to be stored and executed. The ISEF 210 provides the ability to add additional instructions to a set of standard instructions for the processing element 220. The ISEF 210 is a type of software extensible device. In some embodiments, the ISEF 210 comprises a programmable logic device. One example of the ISEF 210 is described in U.S. Application Serial Number 10/404,706 filed on March 31, 2003 and titled "Reconfigurable Instruction Set Computing", which is hereby incorporated by reference.

[00034] The processing element 220 is a processor configured to execute applications. The processing element 220 includes a standard or native instruction set that provides a set of

instructions that the processor element 220 is designed to recognize and execute. These standard instructions are hard-coded into the silicon and cannot be modified. One example of the processing element 220 is an Xtensa processor, from Tensilica, Inc., of Santa Clara, California. One example of the processing element 220 is also described in U.S. Application Serial Number 10/404,706 filed on March 31, 2003 and titled "Reconfigurable Instruction Set Computing."

[00035] The processing element 220 is coupled to an instruction cache 222 and a data cache 224. The instruction cache 222 is a cache configured to store instructions for execution either permanently or temporarily. The data cache 224 is a cache configured to store data either permanently or temporarily. The DP-RAM 230 is also coupled to the processing element. The DP-RAM 230 is a local memory for the processing element 220 that is configured to store data.

[00036] The processor network interface 240 is coupled to the processing element 220. The processor network interface 240 operates as a conduit between the processing element 220 and the network of the array of processor nodes 140. The processor network interface 240 is a communication interface configured to receive data from the processing element 220 and transfer the data to the processor network switch 250 for transport over the network of the array of processor nodes 140. When the processor network interface 240 receives data through the processor network switch 250 from the network of the array of processor nodes 140, the processor network interface 240 transfers the data to the processing element 220. In one embodiment, the processor network interface 240 is coupled directly to the Xtensa Processor Interface (PIF) for the processing element 220, which is an Xtensa processor. In another embodiment, the processor network interface 240 is coupled to the processing element 220 through an AMBA AHB bus. In this embodiment, the attachment to the AMBA AHB bus adds a few more cycles of latency to pass data from the processing element 220 to the processor network interface 240.

[00037] The processor network interface 240 is also coupled to the DP-RAM 230. In one embodiment, the processor network interface 240 is coupled to the DP-RAM 230 through a dedicated port on the processor network interface 240 to transfer software channel data between the processor nodes in the array of processor nodes 140.

[00038] In some embodiments, a programmer can take advantage of the data passing by the processor network interface 240 by using two methods. The first method is by using a memory mapped interface. Using a memory mapped interface, the processing element 220 generates a request to read or write a memory location. The processor network interface 240 then receives the request on the PIF or the AHB bus. The processor network interface 240 then wraps the data as a network packet and transfers the packet onto the transport layer of an OSI layer, which is implemented by the processor network switch 250. When the processor network interface 240 receives a response packet, the processor network interface 240 strips the packet control information and returns the data to the processing element 220 as a transaction on the PIF or AHB bus.

[00039] The second method of data passing is by programming using software channels. A software channel corresponds to a unidirectional stream of data flowing from source to destination. Data injected by the source is transmitted to the destination and delivered in-order. To the programmer, however, channels appear as another memory allocation interface. To send data the programmer allocates a block in memory. When the programmer is done processing the data, the programmer can simply call a subroutine (send) that automatically transmits the data to the destination. The hardware performs the data copying autonomously. This corresponds to a Direct Memory Access (DMA) that copies the data from one memory to another. In one embodiment, the DMA engine transmits the data by encapsulating it into a packet and transmitting it via the network. At the destination, the DMA engine removes the encapsulation

and writes the data into the memory. The hardware ensures that no data is lost and that the source does not inject more data than the destination can process.

[00040] One advantage is that the send and receive software primitives turn access to a stream of data into a memory allocation and they can do this in constant time (i.e. the time to execute the primitive does not depend on the size of the buffer). Converting a stream into a memory buffer is a new programming paradigm.

[00041] In some embodiments, the processor network interface 240 also performs any reads or writes of the DP-RAM 230 that are posted to the AHB bus. When other devices need access to the DP-RAM 230, the processor network interface 240 provides a way to share its dedicated port to the DP-RAM 230.

[00042] The processor network interface 240 is coupled to the processor network switch 250. The processor network switch 250 is a communication interface configured to exchange data with the processor network interface 240. In some embodiments, the processor network switch 250 exchanges data with other network switches in the array of the processor nodes 140. In some embodiments, the processor network switch 250 exchanges data with other devices coupled to the array of the processor nodes 140. In some embodiments, the processor network switch 250 implements the network and link layers of the OSI model and interfaces directly with the physical link.

[00043] FIG. 3 is a block diagram of a processor node 320 with neighboring processor nodes 310 and 330 in an exemplary implementation of the invention. In some embodiments, as a constituent component, the processor node 320 can be "tiled" or combined with other processor nodes, such as west neighbor 310 and/or east neighbor 330 to form a larger, scaled multiprocessor as an array of processor nodes 140 as described above in FIG. 1.

[00044] The processor node 320 comprises an ISEF 321, a processing element ("PE") 322, a data cache 323, an instruction cache 324, a network switch 327, a local memory 326, and

optionally, a cross-bar 325. A north-south (NS) link 328 provides a communication path to the north neighbor processor node (not shown) and/or south neighbor processor node (not shown), and east-west (EW) link 329 provides a communication path to east neighbor processor node 330 and west neighbor processor node 310.

[00045] The data cache 323 and the instruction cache 324 are used, for example, to contain data and instructions, respectively, that the processing element 322 requires to perform its dedicated functionality. These local caches allow data and instructions to be readily accessible to optimize the processing performance. The ISEF 321 can be extensible and customizable such that it can be configured by way of programmable logic to implement new instructions for execution. The new instructions and the ISEF 321 are described in the technology incorporated by reference, such as those described in the U.S. Patent Application entitled "System and Method for Efficiently Mapping Heterogeneous Objects Onto an Array of Heterogeneous Programmable Logic Resources," filed March 31, 2003, under Attorney Docket No. PA2586, which is hereby incorporated by reference.

[00046] In operation, as the processing element 322 executes instructions from instruction cache 323, the processing element 322 can encounter a customized instruction for execution. In this instance, the processing element 322 fetches that customized instruction from the ISEF 321. For example, consider a user generates a "C" program code that yields a new instruction, "Packet Lookup," for a networking application. After compiling this instruction, for example, the compiled new instruction is programmed into the ISEF 321 for execution of this specialized instruction.

[00047] The processor network switch 327 of the processing element 322 is coupled to the NS link 328 and the EW link 329, and is configured to receive and transmit data, instructions and other information. The processor network switch 327 is coupled further to the processing element 322 for communicating data and instructions, for example, to the data cache 323 and the

instruction cache 324, and via the cross-bar 325 for communicating information with the local memory 326. In sum, the processor network switch 327 allows data, instructions and other information to be communicated among an array of processing nodes along the NS link 328 and the EW link 329.

[00048] In some embodiments, in addition to communicating with the other processor nodes 310 and 330, the processor node 320 is adaptable to share resources with other processing nodes in the interest of maximizing utilization of resources. For example, the ISEF 321 is coupled to the processing element 312 of the west neighbor processor node 310. In another example, the cross-bar 325 is coupled to the cross-bar 335.

[00049] The local memory 326 can be configured to receive instructions and/or data, as well as other information that a specific processing element 322 uses to execute its portion of program instructions assigned to that element. For example, in a video compression application, a video image frame can be broken into different blocks. The processor node 320 will receive at least one unique block, such as 16 x 16 pixels, to process in accordance with a video compression scheme. In this instance, the 16 x 16 block of a current frame then will be temporally stored in local memory 326 for performing one or more compression algorithm steps. The local memory 326 can also optionally store a block of pixels from a previous and/or later video frame so as to perform any of the known video compression prediction techniques.

[00050] In some embodiments, the cross-bar 325 is used to provide access to the local memory 326 from the processor network switch 327, a neighboring processing node (e.g., east neighbor 330), and the processing element 322. In one embodiment, XLMI ("Xtensa Local Memory Interface") is the interface used to connect the local memory 326 and the processing element 322.

[00051] In one embodiment, an exemplary size of local memory 326 is 128 kB or 256 kB. In another embodiment, the cross-bar 325 is absent and the local memory 326 has a number of

read/write ports for accessing each of the banks in the local memory 326. That is, at least one read/write port is reserved for interacting with either the processor network switch 327, a local memory of a neighbor processor node (e.g., east neighbor processor node 330), or the processing element 322. In yet another embodiment, the local memory 326 is designed to also be used solely as, or in combination with other functions, a scratch buffer to temporally store intermediate results.

[00052] The local memory 326 is generally adapted to obviate the functionality typically provided by L2 caches known in the art. The local memory 326 is designed to share data with other processor nodes, local memories and/or processing elements in a manner that does not require cache coherency. That is, there is no requirement that a background process ensures that each of the local memories contains the same "coherent" data. But the local memory 326 and its multiprocessor architecture, in one embodiment, are designed to share data with a level of assurance that such shared data is reliable. The local memory 326 can operate as a buffer for buffering incoming and outgoing messages in accordance with a "channel" implementation.

[00053] FIG. 4 depicts a diagram for illustrating communication paths for an array of processing nodes in an exemplary implementation of the invention. The processing node 320 of FIG. 3 is shown in FIG. 4 to have a north, a south, an east and a west neighbor corresponding with a communication link or path between the processing node 320 and its neighbor. These links form a layer of an interconnection network, or a network topology, for communicating information among processor nodes of a grid. This type of arrangement helps provide high-speed neighbor-to-neighbor connections. In one embodiment, each of the communication links can transmit/receive 128 bits wide of data at 500 Mhz (i.e., 8 GB/s), for example.

[00054] Communications between a transmitting processor node 410 of FIG. 4 and the receiving processor node 320 can occur over many paths, where each path is comprised of a number of hops. FIG. 4 shows two exemplary paths, path one ("P1") and path 2 ("P2"), over

which data can traverse. As shown, P1 provides for the shortest path in terms of the number of hops (i.e., 3 hops) while P2 is longer than P1 (e.g., 5 hops). In accordance with one embodiment, an exemplary multiprocessor architecture provides for adaptive routing. That is, communications between specific processor nodes may be prioritized so as to have the shortest number of hops between those processors, the least congested path, and/or any other path that facilitates optimal processing performance. Returning to the example of P1 and P2, if P1 has a longer transmit time because of congestion, for example, then path P2 can be selected to communicate information between nodes 410 and 320. It should be noted that an exemplary programming model in accordance with one embodiment, the number of hops between processing nodes that are designated to perform mission critical functions can be minimized by placing and routing those nodes as close as possible.

[00055] Information (i.e., data, instructions, etc.) is communicated by "message-passing" among arrayed processor nodes. Accordingly, each processing node is associated with a unique node identifier or address ("node ID") by using a packet switched-like network to communicate information between at least two nodes by passing messages including such information. A packet in accordance with one embodiment includes a header and a payload. The processor network switch 327, in some cases, can operate as a "router" as packets are received and either accepted into the processor node 320, or passed on to another switch of another processor node. In one embodiment, the process of message-passing runs in the background without user intervention (e.g., a user need to use these types of explicit message-passing instructions: msg() and/or send ()). Furthermore, by using the exemplary message-passing process, "virtual channels" (i.e., without regard to actual number of physical hops) can be used to pass information (e.g., data) to describe to a user how sub-processes (i.e., processing elements) are interrelated.

Programming Model – FIGS. 5-8

[00056] An exemplary programming model is provided to allow expedient design of a multiprocessor architecture, where such an architecture includes one or more processor nodes. In such a model, the physical hardware is not readily apparent to the programmer, but is represented by abstractions so that a programmer's application can be embodied in a scalable architecture. The programmer can use block diagrams to represent the functionalities of the application, where these diagrams map to an array of processing nodes. In this way, the programmer can design and manufacture one or more chips having four processors or up to 256 processor nodes (i.e., a 16 by 16 grid of nodes), wherein the processors can be configured to use thread-level parallelism, including instruction-level parallelism ("ILP") with ISEF 210 and/or very long instruction set words ("VLIW"). The programmer may also partition the application and identify communications between processors.

[00057] FIG. 5 depicts a block diagram for embedded applications in an exemplary implementation of the invention. FIG. 5 depicts one example of applications for an embedded system that is compiled and loaded into the array of processor nodes 140 of FIG. 1. An embedded application is a computer program or set of instructions for an embedded system. Some examples of embedded applications are a video encoder and a protocol generator.

[00058] Most embedded systems are "static." One example of an embedded system is a set top box. Prior to execution, most properties for the embedded system are known ahead of time. In the set top box example, the largest image size and highest resolution are known before running the embedded applications.

[00059] For these reasons the programming model is "static." In some embodiments, at compile time, the programmer can specify how many tasks will be executing in parallel, how those tasks communicate via channels, what the memory and computation requirements are for

each task, and so on. In some embodiments, the embedded application is recompiled and reloaded to change the system parameters.

[00060] In FIG. 5, the programmer may partition the blocks into smaller blocks for optimal implementation into the array of processor nodes 140. Additionally, certain blocks may be combined into a single processor node in the array of processor nodes 140.

[00061] In one embodiment, each task in a block is programmed in "C" and "Stretch-C," a programming language for software extensible instructions from Stretch, Inc. of Mountain View, CA. The communication bandwidth is then allocated for communications between the tasks. The tasks are then placed onto an array of processor nodes 140. The channels for communications between the processor nodes are routed. The embedded applications depicted in FIG. 5 are then ready for execution.

[00062] FIG. 6 illustrates an example of a conceptual view of an application 600 used in an exemplary programming model of the present invention. The application 600 will be described by way of an example of implementing a video compression encoder in the programming model using "C" programming, a variant thereof (e.g., C++) or any other language. The process 602 represents the overall process of encoding one or more video blocks. The process 602 includes subprocesses 612 and 614, where subprocess 612 represents a motion estimation process (or algorithm in "C") and subprocess 614 represents a variable length coding process (or algorithm in "C"). Each of these subprocesses can have any number of additional layers of subprocesses. As shown, subprocesses 612 and 614 terminate at computational kernels 630 and 632, and 634 and 636, respectively. These computational kernels represent the lowest level of abstraction that includes one or more processing nodes, as determined by the programming model, to design a configurable multiprocessor to perform the user's application.

[00063] FIG. 6 also shows parent-child relations 604, 606, 616, 618, 620 and 622 among the processes, where lower layers of subprocesses are "leaf" processes extending from the root

process 602. Channels 640, 642 and 644 are also shown for channeling data into and out from the processes underlying the kernels. The channels carry the results of each processor node, which is communicated to the next computational kernel for additional processing. For example, stage 608 ("stage 1") represents, in time, the first processes. Then, the results of stage 608 are communicated to stage 610 ("stage 2") for further processing (e.g., variable length coding), which depends upon the first processes' results.

[00064] The following discusses how channels are established, after the computational kernels are done executing. The machine and/or operating system ("OS") configured to operate with the multiprocessor architecture first reserves enough buffering for that communication to succeed. In accordance with the buffering scheme, a sending or a receiving processor node need not involve the OS in the message-passing processes. The OS and/or other machine resources need only be involved in the creation of a channel (to identify, for example, specific processor nodes by node ID and to reserve buffer space), and thereafter is not needed when the code is executed in the computational kernels. Without OS interactions, communication is relatively fast.

[00065] Exemplary pseudo-code (e.g., C language) for the sending processor node and the receiving processor node is described in U.S. Provisional Application Serial Number 60/459,538 titled "Method and Apparatus for an Array of Software Extensible Processors," filed March 31, 2003. In some embodiments, channel creation takes some time relative to other multiprocessor functions because the OS is involved in establishing a channel before any other kind of resource is allocated.

[00066] In one embodiment, in creating child processes and subprocesses, the configuration of parent and child processes and communications between them are assumed to be static. In a reset sequence mode, a number of channels needed to get started is determined before execution of the processes. Each generated child process is associated with a thread ID,

where the use of threads can include some aspects of UNIX®, in accordance with one embodiment. Further, before the last child process is known, the previously generated child processes are associated with execution threads by using a so-called "clone" command as described in Provisional Application Serial Number 60/459,538 titled "Method and Apparatus for an Array of Software Extensible Processors," filed March 31, 2003.

[00067] With execution threads, each of the child processes has access to the same program, data and/or variables, and thus can communicate among themselves. A parent process can update one or more data structures upon which child processes depend. After the data is updated, a flag is cleared so that child processes can be created (i.e., "wait" is set before creating children). When the wait flag is cleared to zero, the created children wake up and will start executing their code. At that point, they know whether the data structure is up to date, and they can rely on the values. Moreover, a child process might recognize that it is designated as "thread ID 3" and that it is required to communicate with "thread ID 5." These children are executing in the same address space, and see the same variables as well as program, etc. After the children begin executing with an "exec" command, the children separate from the parent (e.g., and are operating independent from the OS). After the channels have been created and the children have been separated from their parents, the parent processes can signal to the OS that placement and routing processes can proceed, because the parent-child and child-child communications have been established and are not likely to change in the future.

[00068] For many embedded applications, it is very useful to be able to guarantee the communication bandwidth between two nodes. In the case of video compression (or decompression), for example, it is useful to be able to guarantee the bandwidth required for the uncompressed video. If the machine cannot deliver the required bandwidth, the quality of the image will suffer.

[00069] In some embodiments, a user may place bandwidth requirements in the communication channels. The hardware will then guarantee that the amount of bandwidth is always available. The bandwidth is then reserved and not available for other uses.

[00070] In one embodiment, the hardware guarantees communication bandwidth by using time division multiplexing (TDM). This is similar to the bandwidth allocation used in the switched phone network but has not been applied as the communication medium for a multi-processor network. In TDM, time is first divided into "slots" (or timeslots) with N timeslots per period. During each timeslot, each link is pre-allocated to a particular connection (channel). Furthermore, the connections of each link are pre-determined to enable the connections of links across time.

[00071] FIG. 7 depicts a diagram of an array of processor nodes using time division multiplexing in an exemplary implementation of the invention. In FIG. 7, a channel carries data from node 700 to node 704. The data first travels at timeslot 0 from node 700 to node 701 via link 710. At timeslot 1, the switch (not shown) at node 701 takes one cycle to pass the data on link 711. At timeslot 2, the link 712 then carries the data to node 703. Finally, at timeslot 3, the link 713 carries the data travels to node 704. Thus, a connection from a source node to a destination node can be built by pre-allocating link and switch bandwidth to each connection between the intervening nodes. In some embodiments where applications are static, the place and route problem is solved only once.

[00072] Routing the set of channels in an application consists of a maze router in three dimensions. Each level of the maze corresponds to a particular timeslot. Data moves vertically when it arrives at a switch and is held for a cycle. In some embodiments, no control information is routed along with the data. In these embodiments, assuming that network has been setup correctly, the data simply arrives at the destination node. This requires that all nodes be synchronized in time.

[00073] TDM can be implemented using a table with one entry per timeslot. This table indicates the connections that should be enabled in the crossbar (i.e. which egress port to use for each ingress port). A counter keeps track of the current timeslot and is used to select an entry in the table.

[00074] TDM can be thought of as virtually partitioning the bandwidth of the link into individual "virtual wires." If there are 128 timeslots in a period and 128-bits in each link, then each timeslot corresponds to a dedicated 1-bit virtual wire.

[00075] A related, alternative implementation of guaranteed bandwidth is the concept of bundling, which is also called spatial division multiplexing. Bundling is the spatial equivalent of time division multiplexing. That is, rather than allocating bandwidth by dividing time it allocates bandwidth by dividing the individual wires in the link.

[00076] If each link consists of 128 wires, the link can be divided into 128 individual "bundles." This concept is referred to as bundles because more than one wire can be used for each connection (for example, some connections could have 4 or 8 wires). Each wire is then dedicated to a particular connection and cannot be shared in time. One advantage of bundles over TDM is that global time synchronization is not required. Data will simply travel down the wire and eventually be received at the destination.

[00077] FIG. 8 depicts a diagram of an array of processor nodes using bundling in an exemplary implementation of the invention. FIG. 8 shows a simple example of an array connected by bundles (where one bundle 810-813 shown in bold is used to connect node 800 and node 804). With bundles, the complexity of the switch increases since it must be able to switch every bit individually. Furthermore, more hardware is required at the network interface to slowly insert a 64 or 128-bit quantity into a 1, 2, or 4-bit "wire." This limits how many connections can be active at once.

Board Level Array of Processing Nodes – FIGS. 9-10D

[00078] In some embodiments, each of the processor nodes is on a separate chip. The chips can be configured together on a board to form the array of processing nodes 140 as depicted in FIG. 1.

[00079] FIG. 9 depicts a diagram of a software extensible processor chip 900 in an exemplary implementation of the invention. The software extensible processor chip 900 includes multiplexer/demultiplexers (mux/demux) 912, 922, 932, and 942, standard I/O interfaces 914, 924, 934, and 944, array interface modules (AIM) 916, 926, 936, and 946, and a software extensible processor tile 950. The software extensible processor tile 950 is a processor node 200 as described in FIG. 2. The software extensible processor tile 950 is coupled to the standard I/O interfaces 914, 924, 934, and 944 and AIMs 916, 926, 936, and 946.

[00080] The standard I/O interface 914 is an interface configured to handle standard I/O processing between chips. Some examples of the standard I/O interfaces 914 are a peripheral component interconnect (PCI) interface, a DDR interface, and a universal asynchronous receiver/transmitter (UART) circuitry. The standard I/O interface 914 is coupled between the software extensible processor tile 950 and the mux/demux 912. The other standard I/O interfaces 924, 934, and 944 have similar operations and configurations as the standard I/O interface 914.

[00081] The AIM 916 is an interface or circuitry configured to handle a protocol for communication between software extensible processor chips 900. In some embodiments where processing nodes are on the same chip, the network switches can be coupled through wires. In other embodiments where processing nodes are on different chips, a different protocol for handling off-chip communications with other processing nodes is needed. The AIM 916 handles these off-chip communications with other software extensible processor chips to provide a

physical layer to couple the software extensible processor chips together. In some embodiments, the protocol that the AIM 916 uses handles buffering between the processor chips. In some embodiments, the protocol that the AIM 916 uses accounts for latency problems for communications that go off-chip. The AIM 916 is coupled between the software extensible processor tile 950 and the mux/demux 912. The other AIMs 926, 936, and 946 have similar operations and configurations as the AIM 916.

[00082] The mux/demux 912 is configured to multiplex or demultiplex between the link 918 and the standard I/O interface 914 and the AIM 916. The mux/demux 912 is also configured to select between the standard I/O interface 914 and the AIM 916 based on the type of neighboring device that the link 918 is coupled to. For example, if the neighboring device is an I/O device, then the mux/demux 912 is configured to select the standard I/O interface 914. If the neighboring device is another software extensible processor chip, then the mux/demux 912 is configured to select the AIM 916. In some embodiments, software or an external pins determines the selection. The other mux/demuxes 922, 932, and 942 have similar operations and configurations as the mux/demux 912.

[00083] FIGS. 10A-10D illustrate different configurations for software extensible processor chips. In these embodiments, each software extensible processor chip has four interfaces for communications with a north, east, south, and west neighbor. The solid bar interfaces represent standard I/O interfaces. The hashed bar interfaces with diagonal lines indicate the AIMs. These representations of solid bars and hashed lines are merely to illustrate which interface the respective mux/demux has selected for communications with the neighboring device.

[00084] FIG. 10A depicts a diagram of two software extensible processor chips 1010 and 1020 in an exemplary implementation of the invention. The standard I/O interfaces 1012, 1014, and 1018 are depicted as solid bars. The AIM 1016 is depicted as a hashed bar. The software

extensible processor chip 1010 comprises a standard I/O interface 1012 for a neighboring device to the west, a standard I/O interface 1014 for a neighboring device to the north, an AIM 1016 for a neighboring device to the west, and a standard I/O interface 1018 for a neighboring device to the south. The two software extensible processor chips 1010 and 1020 communicate with each other through the AIMs 1016 and 1022.

[00085] FIG. 10B depicts a diagram of four software extensible processor chips 1030, 1035, 1040 and 1045 in an exemplary implementation of the invention. The software extensible processor chips 1030, 1035, 1040 and 1045 are connected to each other forming a line. The software extensible processor chips 1030, 1035, 1040 and 1045 communicate with each other through their respective AIMs. For example, the software extensible processor chip 1030 communicates with the software extensible processor chip 1035 through the AIMs 1031 and 1036. The software extensible processor chip 1035 communicates with the software extensible processor chip 1040 through the AIMs 1037 and 1041.

[00086] FIG. 10C depicts a diagram of a three by three array 1050 of software extensible processor chips in an exemplary implementation of the invention.

[00087] FIG. 10D depicts a diagram of two software extensible processor chips 1060 and 1070 with a software extensible system on a chip (SOC) 1080 in an exemplary implementation of the invention. The software extensible processor chip 1060 communicates with the software extensible processor chip 1070 through the AIMs 1061 and 1071. The software extensible processor chip 1070 communicates with the software extensible SOC 1070 through the standard I/O interfaces 1072 and 1081. The software extensible SOC 1080 comprises an array of processor nodes.

Routing Algorithm – FIGS. 11A-12

[00088] Microprocessor networks are usually built using regular topologies, for example, an array or a cube. Some embodiments support non-rectangular configurations. FIGS. 11A-D depict diagrams of four non-rectangular configurations in exemplary implementations of the invention.

[00089] There are two main reasons to support non-rectangular configurations. The first reason is for sharing the pins between the AIMs (used to build the array) and standard I/O interfaces (DDR, PCI, etc.). Since particular nodes may need to be attached to some standard I/O, an assumption that the nodes are placed in a regular array cannot always be made. Furthermore, the communication graph for the application usually are not rectangular and should match the configuration of processor nodes.

[00090] FIG. 12 depicts a routing algorithm in an exemplary implementation of the invention. FIG. 12 depicts the routing algorithm consisting of the following steps. In this embodiment, the steps are executed in priority order as listed. In process 1210, a comparison is made between the destination network address of the packet and the processor node's network address. If destination network address of the packet matches the processor node's network address, then the processor node is the destination for the packet and the packet is processed.

[00091] If the packet is statically routed, the packet ID is used to index into the static routing table in process 1220. In one embodiment, the ID field is 4 bits and the static routing table consists of 16 entries. Each entry has a two bit field that indicates the egress port for the packet.

[00092] In steps 1230 and 1240, the destination address is subtracted from the node address for the X and Y components, respectively. If the difference is in the range [-2, 2] (for both horizontal [x] and vertical [y] components), then the difference is used to index into a 25-entry table called the near table in process 1250. Each table entry holds a 2-bit field that indicates

the egress port of the packet. The outgoing port is determined given the relative location of the destination.

[00093] In steps 1260 and 1270, the destination address is subtracted from the node address for the X and Y components, respectively. 4. In process 1280, the sign of the difference (both horizontal and vertical) is used to index into a 4 entry table called the far table that indicates the egress port of the packet.

[00094] This routing algorithm advantageously supports statically routed packets. Also, this routing algorithm supports arbitrary (i.e. non-minimal) routes with the limitation that each route can traverse each node only once. This routing algorithm supports table driven routing. In one example, the support is for up to 25-node machines. Since the routing is table driven, the algorithm can support any arbitrary topology. The routing algorithm can also support large-scale machines such as up to 256 nodes. For large machines, packet routing is broken into two steps. While still far-away from the destination, the packet is routed based on the quadrant the destination is in (i.e. upper-left, upper-right, lower-left, or lower-right). Once close to the destination, the packet is routed using the near-table entries.

[00095] The above description is illustrative and not restrictive. Many variations of the invention will become apparent to those of skill in the art upon review of this disclosure. The scope of the invention should, therefore, be determined not with reference to the above description, but instead should be determined with reference to the appended claims along with their full scope of equivalents.